by
Ray Duncan

Power Programming

Operator and Function Overloading In C and C++

■ The fundamental difference between C and C++—and the feature that makes object-oriented programming possible—is that C++ not only allows the programmer to overload functions and operators, it actively encourages doing so.

In a pure object-oriented programming language (OOPL) like Smalltalk or Actor, the well-defined entity that we oldtimers think of as an application program really doesn't exist. That is to say, the source code for an application does not reside in any single location, so there is no way to view it as a whole or print it out. Instead, the source code is distributed across myriad definitions of *classes,* some of which are fundamental to the language and entirely designed by the language vendor, some of which are derived or ''subclassed'' by the application programmer from code written by the vendor, and some of which are unique to the application. All of these different classes are linked together in a treelike hierarchy whose root is a ''superclass'' that owns everything and does nothing.

When you peek inside a class definition, you typically find declarations of private data items and of some procedures or routines called (in OOPL-speak) *methods.* Each method is triggered by a specific *message*—a value that is magically transported from one place to another by the programming milieu. A method usually responds to a message by inspecting or changing one of the data items and often by sending a message in return. If the class definition doesn't provide a method for a particular message, the message is successively ''kicked upstairs'' through the network of ancestor classes until it can be processed.

But a class definition is only a template, like a struct{} declaration in the C language. The class must be ''instantiated'' as one or more objects before it becomes a functional part of the application, just as a struct{} must be instantiated as storage before it can be used. An object is an object (as opposed to a variable or procedure in a traditional programming language) because it can be addressed by a unique name and has both content and behavior. The content, or state, of the object is determined by the values of its private data items. The behavior of the object is determined both by the values of the private data items and by the program code that constitutes the object's methods (of course, this code actually resides in the class definition or in one of the class's ancestors and is shared by all the objects that are members of that class).

In this brief recapitulation, we see all three of the classic hallmarks of a true OOPL—polymorphism, inheritance, and encapsulation—and we also see that they follow naturally from the fundamental concepts of objects and classes. The mere fact that a class can contain more than one method and thus respond to more than one type of message yields polymorphism. The ability of a class to exploit methods defined in its ancestors (by the simple strategy of passing onward a message it doesn't know how to handle) is the very essence of inheritance. And the ''black box'' nature of a class and its objects—that messages go in and messages come out, but an object's methods and private data are not directly accessible by any other object—is the epitome of encapsulation.

Although the pure object-oriented programming languages implement polymorphism, inheritance, and encapsulation in an exceedingly elegant manner, this elegance has brought them more critical acclaim than market share, and language vendors can't take critical acclaim to the bank. No doubt this was one of the primary reasons Bjarne Stroustrup decided to graft object-oriented capabilities onto the existing C language and ride piggyback on its rapidly growing popularity, instead of inventing yet another OOPL of his own. We are told by C++'s partisans that it is a multiparadigm language; in other words, it can be used both for traditional, procedural programming in the C style and for object-oriented programming. This claim ought to set off an alarm bell or two in your head immediately, but more about that later in this article. In the meantime, let's venture a little further into the weird and wonderful world of C++ by looking at operator and function overloading, which is the foundation of polymorphism in C++.

OVERLOADING IN C

When we say that an operator or function name is *overloaded,* we mean that the symbol can be used in two semantically different ways, and the compiler will figure out the proper code to generate. The C language has a certain amount of operator overloading built-in, although this is something that most C programmers never even think about. Consider, for example,

the following snippet of C code:

```
int i1, i2, i3;
long l1, l2, l3
double f1, f2, f3

i1 = i2 + i3;
l1 = l2 + l3;
f1 = f2 + f3;
```

When the compiler runs into the + operator, which is overloaded, the code that it generates is determined by the data types of the items being added. The following line

```
i1 = i2 + i3;
```

is translated into code like the following:

```
mov   ax,i2
add   ax,i3
mov   i1,ax
```

TRYVECS.CPP 1 OF 2

```
// TRYVECS.CPP - Try Overloaded + Operator for Vector Data Type
// Compile with Borland C++ 2.0
// Copyright (C) 1991 Ziff Davis Communications
// PC Magazine * Ray Duncan April 1991

// Note: all directions for input and output are in degrees,
// but calculations are carried out internally in radians.

#include <math.h>
#include <iostream.h>

const double pi = 3.141592654;              // constant Pi

                                            // local function prototypes
double deg2rad(double);                     // convert degrees to radians
double rad2deg(double);                     // convert radians to degrees

struct VECTOR {                             // vector data type
    double magnitude;
    double direction; } ;

VECTOR operator + (VECTOR A, VECTOR B);     // vector operator prototype

main()
{
    VECTOR A, B, C;                         // instantiate 3 vectors

    cout << "\nAdd two vectors.";
    cout << "\nNote: directions are entered in degrees!\n\n";

    cout << "Enter Vector A magnitude: ";   // prompt for Vector A
    cin  >> A.magnitude;
    cout << "Enter Vector A direction: ";
    cin  >> A.direction;

    cout << "Enter Vector B magnitude: ";   // prompt for Vector B
    cin  >> B.magnitude;
    cout << "Enter Vector B direction: ";
    cin  >> B.direction;

    A.direction = deg2rad(A.direction);     // convert degrees to radians
    B.direction = deg2rad(B.direction);

    C = A + B;                              // add the vectors

    C.direction = rad2deg(C.direction);     // radians to degrees

    cout << "\nVector result: "              // display the result
         << "  magnitude = " << C.magnitude
         << "  direction = " << C.direction << "\n" ;
}

VECTOR operator + (VECTOR A, VECTOR B)      // vector addition operator
{
    VECTOR temp;                            // scratch storage
    double angle;                           // scratch storage
```

Figure 1: TRYVECS.CPP is an interactive program written in Borland C++ 2.0 that demonstrates operator overloading. TRYVECS prompts you for two vectors, adds them together, and displays the resulting vector.

Similarly, the line

```
l1 = l2 + l3;
```

is translated to code of this sort:

```
mov   ax,word ptr l2
mov   dx,word ptr l2+2
add   ax,word ptr l3
adc   dx,word ptr l3+2
mov   word ptr l1,ax
mov   word ptr l1+2,dx
```

And the line

```
f1 = f2 + f3
```

results in this code:

```
fld   qword   ptr f2
fadd  qword   ptr f3
fstp  qword   ptr f1
```

If you mix data types on an overloaded operator, the compiler looks even smarter; it will automatically produce code that coerces data types as necessary and then adds the results. For example, when faced with this mixture of three data types

```
f1 = i1 + l1
```

the compiler forges ahead anyway and produces

```
mov   ax,i1
cwd
add   ax,word ptr l1
adc   dx,word ptr l1+2
mov   word ptr temp,ax
mov   word ptr temp+2,dx
fild dword ptr temp
fstp qword ptr f1
```

However, the overloading of C's + operator is more limited than it appears at first. It would be nice to be able to write statements like

```
char * string1;
char * string2 = "ABC";
char * string3 = "DEF";

string1 = string2 + string3
```

But this interpretation of the + operator was not hardwired into the compiler, so you just get the error message "Invalid pointer addition." The compiler isn't even smart enough to handle perfectly reasonable extrapolations of operations it al-

TRYVECS.CPP 2 OF 2

```
    angle = B.direction - A.direction;      // find angle between vectors

    if (angle == pi)                        // special handling to avoid
    {                                       // overflow for angle=180
      temp.magnitude = fabs(A.magnitude - B.magnitude);
      temp.direction = A.magnitude>B.magnitude ? A.direction : B.direction;
      return temp;
    }

    if ((A.magnitude == 0) && (B.magnitude == 0))
    {
      temp.magnitude = 0;                   // special handling to
      temp.direction = 0;                   // avoid divide by zero
      return temp;                          // if both magnitudes = 0
    }

    temp.magnitude = sqrt(                  // find magnitude of result
                     (A.magnitude * A.magnitude) +
                     (B.magnitude * B.magnitude) +
                     (2 * A.magnitude * B.magnitude * cos(angle)));

    temp.direction = A.direction +          // find direction of result
                     asin(((B.magnitude * sin(angle))/temp.magnitude));

    return temp;                            // return resultant vector
}


double deg2rad(double degrees)              // convert degrees to radians
{
    return ((degrees * 2 * pi)/360);
}


double rad2deg(double radians)              // convert radians to degrees
{
    return ((radians * 360)/(2 * pi));
}
```

ready knows, such as the following:

```
struct point {
   int x;
   int y;
   int color; } ;


main()
{
   // declare 3 points
   struct point A, B, C;
   // add the points
   A = B + C;
}
```

This code makes the compiler choke, with the error message "Illegal structure operation." And since all overloading in C is hardwired by the compiler's author, and each instance of overloading is handled as a special case with magical code generation, there's no way for the applications programmer to extend the compiler by defining new capabilities for an operator, let alone new operators.

As for overloading function names, forget it; a C compiler can't handle that. If you want functions that do symmetric things with different data types, you must give each function a distinct name. For example, it would be convenient to have a single display function, print(), that could identify the data type passed to it and convert the value to ASCII accordingly. But in C, you can't accomplish this in any natural way; you must code a set of parallel functions in this way:

```
void iprint(int i)
{
     printf("%d", i);
}

void lprint(long l)
{
     printf("%ld", l);
}

void fprint(double f)
{
     printf("%f", f);
}
```

The burden is on you to remember the proper function name for the type of data you want to display.

OVERLOADING IN C++

To me, the fundamental difference between C and C++—and the feature that makes object-oriented programming possible in C++—is C++'s treatment of overloading. C++ not only allows the programmer to overload functions and operators, it actively encourages doing so.

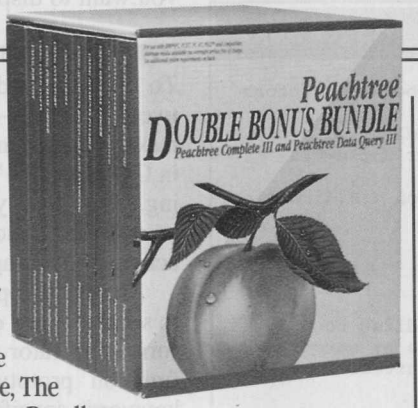
An intrinsic operator can be overloaded to service a new data type by providing a formal operator prototype (much like a function prototype) with an operator keyword, specifying how the operator behaves by providing a routine to implement the operator's new behavior, and using the operator in the "expected" manner. For example, if we want to overload the + operator so that it can be used to add vectors, a skeleton of our program would look something like this:

```
// declare vector data type
struct VECTOR {
    double magnitude;
    double direction; } ;

// vector addition operator
 prototype
VECTOR operator + (VECTOR A,
 VECTOR B);

main()
{
  // create 3 vectors
    VECTOR A, B, C;
  // add the vectors
    C = A + B;
}

// this routine implements vector
 addition
VECTOR operator + (VECTOR A,
 VECTOR B)
{
  ...  // vector-specific code here
}
```

To illustrate this concept in more detail, I've provided the example program TRYVECS.CPP in Figure 1. TRYVECS implements the vector addition operator in the context of a little interactive demonstration program that prompts you to enter two vectors, adds them, and then displays the resulting vector. You can compile TRYVECS with Borland's C++ 2.0 package. A sample session with

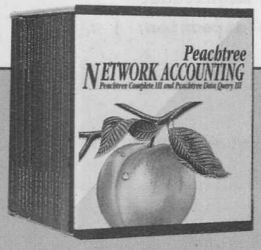
PEACHTREE ACCOUNTING MAKES

When you're running your own business, every financial decision is an important one. Why not make sure you are getting the most out of your business dollars by using the best value in accounting software, The Peachtree Double Bonus Bundle■



"Rivals packages costing 10 times as much...a joy to use..."
–PC Resource

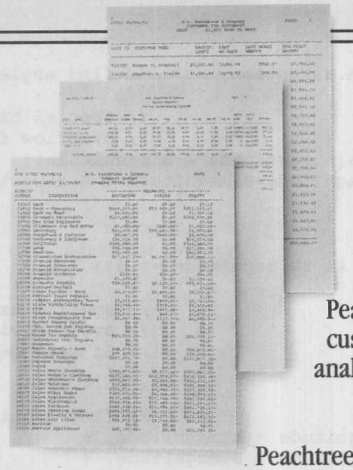
"This package is truly a bargain and is ideal for a small to mid-sized company."
–Computers in Accounting



Give every workstation the power of Peachtree with Peachtree Network Accounting!

Now your entire staff can access the full-featured accounting power of Peachtree Complete III plus the custom reporting capabilities of Peachtree Data Query III in one package–Peachtree Network Accounting. Not only will your hardware costs be reduced by sharing hard disks and printers, but your authorized users will have faster, easier access to shared accounting information. Available now for use with Novell®, LANtastic™ and other NETBIOS compatible LANs.

$695



Peachtree has combined two high-powered programs into one value-packed bundle–Peachtree Complete III, a comprehensive accounting system, and Peachtree Data Query III, a custom reporting and analysis tool■

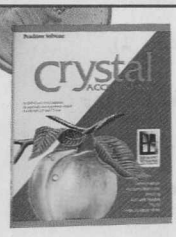
Peachtree Complete III has nine powerful modules, each loaded with features you'd expect to find in much higher priced packages:

- *General Ledger*
- *Accounts Payable*
- *Accounts Receivable*
- *Payroll*
- *Inventory*
- *Invoicing*
- *Purchase Order*
- *Fixed Assets*
- *Job Cost*

The management reporting capabilities of Peachtree Data Query III provide you with personalized views of the accounting data in your Peachtree Complete III files. Whether you choose one of the 50 pre-defined reports or customize your own, PDQ lets you get to the meaning behind the numbers with historical reports and complex analyses to give you a more complete picture of your business finances■

"The reports will impress you, your accountant, your banker and the IRS... a wonderful tool to help get your business on the right track, and keep it that way in the coming years."
–Journal of the Chicago Computer Society

There's a Peachtree Accounting Product for Every Small Business.



Crystal Accounting!™ The Clear Choice for Windows Accounting. Includes G/L, A/R, Invoicing, A/P, Payroll and Financial Report Writer. **$495**

Peachtree Complete III!® The Biggest Value In Small-Business Accounting. Includes G/L, A/P, A/R, Invoicing, Payroll, Inventory, Purchase Order, Fixed Assets and Job Cost. *Winner of the 1990 Computer Shopper Best Buy Award.* **$199**

Peachtree Client Write-Up. The Accountant's Answer to Increase Productivity. Will interface to Peachtree Complete III, Crystal Accounting and other leading accounting systems. **$995**

EVERY PENNY COUNT.

With PDQ, you can even read and report on data from your Lotus 1-2-3® spreadsheets, and export data to Lotus 1-2-3, dBASE® and other popular applications with no re-keying of data and no risk to your files.

Best of all, you won't sacrifice ease of use to get the power and performance your business needs. There's an on-line tutorial in each module and an Accounting Primer, so you can learn accounting as you set up your files. Context sensitive HELP is on-line at all times. Peachtree provides a thorough reference library, including a quick-start installation guide to assure that you'll be up and running quickly. Errors are handled with understandable, plain English messages and suggestions■

"I have installed and set up Peachtree for many small businesses. After the first 30 days, I become the Maytag® Repairman."

–Karen R. Gooding
Economy Bookkeeping, Hilton Head Island, SC.

FEATURES

General Ledger
- Repeating and reversing journal entries.
- One to 13 user-defined fiscal periods.
- Comparative financial statements.

Accounts Payable
- Partial payment of invoices.
- Cash requirements forecasting.
- Check printing from 9 different bank accounts.
- Prints miscellaneous and interest 1099s and handles magnetic media.

Accounts Receivable/Invoicing
- Open item or balance forward customers.
- Supports partial payments.
- Multiple "Ship To" addresses per customer.

Purchase Order
- Issues purchase orders for either stocked or non-stocked items.
- Inventory update upon receipt of items.
- Ability to generate A/P invoice from purchase order.

Payroll
- Built-in federal, state, city and county tax tables for all 50 states.
- Automatic payroll processing supporting hourly, salaried, commission or draw-against-commission pay types.
- Supports Cafeteria and 401K Plans.
- Optional Peachtree Complete Tax Service available.

Inventory
- Supports LIFO, FIFO, Specific Unit, standard and average costing methods.
- Optional serial number tracking.
- Supports assembly and disassembly capabilities.

Job Cost
- Tracks costs and profitability on a job-by-job basis.
- Interfaces with A/P, A/R and Payroll.

Fixed Assets
- Handles 13 methods of depreciation.
- Handles short taxable year.
- Mid-quarter convention.

Peachtree Data Query III
- Transfer accounting information in ASCII or WKS formats to spreadsheet and database programs such as dBASE and Lotus 1-2-3.
- Create your own custom reports and save the formats for future use.
- Print stored reports directly from your Peachtree Complete III menu.

"I would highly recommend Peachtree... a peach of a program."

–New Jersey PC User Group News

In a recent survey by *Accounting Today* magazine, Peachtree was recommended by accountants almost twice as often as the nearest competitor. When *Reseller Management* asked resellers which accounting package they recommend, Peachtree came out on top again, with 25% of resellers recommending it.

Most importantly, users recommend Peachtree! Based on reader response, *Computer Shopper* named Peachtree Software the Best Buy in Accounting Software for 1990. More than 400,000 small businesses trust their accounting to Peachtree■

"...How good is technical support? I can honestly say it is excellent!"

–Jack J. Neymark
Jack J. Neymark Accounting Services

Convenient technical support may be purchased from Authorized Support Centers in over 60 cities across the country or through prepaid support contracts directly from Peachtree. We stand behind our products with a 60-day money back guarantee when you buy directly from Peachtree. If you aren't satisfied with your Peachtree product, simply return it within 60 days for a prompt refund (less a restocking fee.)■

Call now to order

or for the name of a dealer in your area

1-800-247-3224

or 404-564-5800 or to order by Fax 404-564-5888

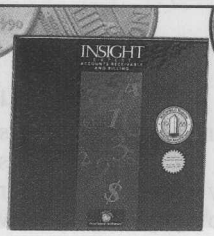
$298 *Add $12.50 Shipping*

In Georgia, add applicable sales tax
Telephone number required on all orders

Peachtree Software

Hardware Specifications: Requires PC/MS-DOS version 3.0 or higher (DOS 3.1 required for Network Accounting) with 640k of memory and a hard disk. For use with IBM® PC, PC XT, PC AT, Personal System/2™ and compatibles. Alternate media optionally available. Not copy protected. Product names referenced are trademarks and registered trademarks of their respective manufacturers.

atOnce!® Macintosh Accounting as Easy as it Gets. Includes G/L, A/R and Billing, A/P and Payroll. *Winner of the MacUser Eddy Award and the Macworld World Class Award.* **$395**



INSIGHT EXPERT.™ The Power to See Beyond the Numbers for Macintosh Users. Includes G/L, A/R and Billing, A/P, Inventory and Payroll. *Winner of the MacUser Eddy Award and the Macworld World Class Award.* **$695 per module.**

CIRCLE 358 ON READER SERVICE CARD

PCM

A TRYVECS EXAMPLE

```
C>tryvecs

Add two vectors.
Note: directions are entered in degrees!

Enter Vector A magnitude: 100
Enter Vector A direction: 45
Enter Vector B magnitude: 100
Enter Vector B direction: 135

Vector result:  magnitude = 141.421356  direction = 90

C>
```

Figure 2: A sample session with the TRYVECS demonstration program.

TRYVECS is shown in Figure 2 above.

Functions are even easier to overload. You don't have to use a special keyword as you do with operators; you just create the different incarnations of the function with the appropriate prototypes and function declarations. To implement the "smart" print function we mentioned earlier, we could just write:

```
// function prototypes
void print(int);
void print(long);
void print(double);

.
.
.

void print(int i)
{
     printf("%d", i);
}

void print(long l)
{
     printf("%ld", l);
}

void print(double f)
{
     printf("%f", f);
}
```

We can then invoke print() with any of the three specified data types, and it will perform correctly:

```
print(1);
print(2000000L);
print(3.14159);
```

If you call an overloaded function with a parameter whose type matches none of the function declarations, C++ will find the function that matches most closely and perform the necessary data conversions.

There are surprisingly few constraints on the overloading of functions. The most important requirement is that overloaded functions not differ only in the type of the value they return; they must differ in at least one of their formal parameters. This is because both the C and C++ programming languages allow the return values of any function to be ignored at the caller's discretion. For example, it would not be legal to overload our print() operator as follows:

```
// function prototypes
void print(int);
int print(int);

.
.
.

void print(int i)
{
    printf("%d", i);
}

int print(int i)
{
    printf("%d", i);
    return i;
}
```

The restrictions on overloading of operators are more extensive:

- The operator must already exist in the language (you can't dream up completely new operators).
- You can't define a new action for an intrinsic operator on a native C++ data type; for instance, you can't change the way the + operator works on two **int**s.
- You can't change the precedence of intrinsic operators when you overload them.
- You can't overload operators with different prefix and postfix actions (despite the fact that C++ itself has such operators).
- You are not allowed to overload the operators **.**, **.***, **::**, and **?:** (you may not have heard of :: yet, but you will soon enough).

By this time, I'm sure you can see that operator and function overloading alone have the potential to improve the readability of programs immensely. If Stroustrup had limited his redesign of the C language to the syntactic improvements we discussed in the last column and the overloading that we've covered here, he would have given us a far more powerful, somewhat extensible language without (in my opinion) any significant flaws. But the result would still have been a traditional procedural language—a sort of Super-C. It was Stroustrup's further addition of notations for object declaration and inheritance that turned C++ into a true OOPL and incidentally into one of the most grotesque and cryptic programming languages man has ever created, as we'll see in the next two installments of this column.

Stroustrup's addition of notations for object declaration and inheritance turned C++ into a true—and cryptic—OOPL.

THE IN-BOX

Please send your questions, comments, and suggestions to me at any of the following e-mail addresses:
PC MagNet: 72241,52
MCI Mail: rduncan
BIX: rduncan